Introduction to OCAML

OCAML[1] is an evolution of the ML programming language[2]. It focuses on bridging the gap between the more esoteric ML language with its functional programming roots and more modern practices like objects.

OCAML may not be as widespread as other languages, but it has found a number of modern uses[3], perhaps most notably by the Jane Street[4] trading firm.

Features of OCAML

Some key features of OCAML

- **Static Typing**: OCAML is a strongly typed language. Every variable/value/expression has a type, at compile type.
- **Strong Typing**: There are, with few exceptions, no automatic coersions. For example a boolean would not be automatically turned into an integer, and a non-zero number would not be automatically treated as a boolean in a conditional.
- **Type Inference**: In almost all cases, you do not need to specify the type of a value/variable/expression. The system automatically figures it out via a fascinating process called *type inference*.
- **Strong and Expressive Type system**: With the use of user-defined types, pattern matching and tuple formation, and polymorphic types, OCAML offers a very rich level of expressiveness and compile-time guarantees to programmers.
- **Functional Programming**: OCAML espouses functional programming paradigms. Mutable values are discouraged, though possible, and there is extensive support for higher-order-functions, as well as for using functions as first order values, i.e. as return values, arguments or local variables in other functions (Javascript is a popular language that shares that feature).
- **All functions take exactly one argument**: This is probably very surprising, but we will talk more about it later on. In any case it is a fascinating feature.
- **Everything is an expression**: Expressions return values, statements do not. Essentially everything in OCAML is an expression. Even an if-then-else block for example.
- **Automatic Memory management**. Like most non-C/C++ languages, OCAML does not require manual memory allocation/deallocation.
- **Objects**: OCAML also supports object-oriented programming. We will not however examine this nature of the language much.

[1] http://ocaml.org/
[2] https://en.wikipedia.org/wiki/ML_(programming_language)
[3] http://ocaml.org/learn/success.html
[4] https://www.janestreet.com/

Interactive Sessions with utop

The easiest way to get going with OCAML is to use the utop command line tool. We will need to start by installing it. Open up a terminal, and you should find a tool called opam. You will need to do the following:

```
opam init --dot-profile=~/.bashrc
```

This will create your .bashrc file if it does not already exist, and add some necessary lines to it. OPAM is a package manager for OCAML. We will now use it to install utop. You would want to do:

```
opam install utop
```

You may need to run opam update first.

Now you should be able to type utop. You should be greeted by a prompt as well as some other useful information, including available functions as you type. You can use the up/down arrows to go back and forth through the history of previously typed commands.

Let's take it for a spin. Type:

```
4 + 5;;
```

You should see a reply that tells you that the result value was 9 and it was of type int. Note that you have to use 2 semicolons to indicate that you finished typing the command and you want OCAML to run it. You do not need that when working from a file.

Let's try another one:

```
2.2 + 3;;
```

Oops, now you encounter your first error, and in fact it is an error you would probably not have expected. in OCAML there is no automatic coersion. You have told it to add something to 3, which is an integer, but you gave it a floating point number (a value of type float in OCAML). OCAML will not let you do that: You would need to convert 3 to a float first. You can do that by either typing 3.0 instead, or using the function float_of_int:

```
float_of_int 3;;
```

You will see a couple of things here. First, we called a function, but we did not include parentheses around the argument. In fact you almost never have to. So what we wrote there would in other languages have looked more like float_of_int(3). Not so in OCAML. OCAML knows that float_of_int is a function, and hence treats what follows as its argument.

Second, the result is now of type float, and it has value 3., the dot indicating it is a float. So this function turned our integer into a float.

Okay now let's try to put that in place:

```
2.2 + float_of_int 3;;
```

You will see the error again, and at this point it seems probably even more mysterious. This is because the addition operator + is actually expecting integers. We need to use a different symbol, +., in order to add floats:

```
float_of_int 3 +. 2.2;;
```

Note that function application "binds" quite strongly, it has high precedence. The function application on 3 happens first, and the addition operator acts on the result of that.

Before we move on, let us talk about the other basic types. You can find many of the standard operators in what is known as the Pervasives module[5], which gets automatically loaded in. We will probably talk about modules later on.

Here is a short list of standard OCAML types:

int signed, 31/63-bit depending on architecture

float IEEE double precision, equivalent to most language's "double"

bool Boolean. Values `true`, `false`.

char 8-bit character. Does not support Unicode, UTF-8. Write in single quotes: `'a'`.

string Write in double quotes: `"a string"`.

unit Takes a single possible value, written `()`. Typically used to indicate that the function needs no input, or returns no meaningful result. C would typically use `void` for this, Javascript would use `null`.

product types In OCAML you can easily "pair up" other types of values. For instance the value `(2, true)` is of the product type `int * bool` (try it out!). These kinds of values are called *tuples*. They can be used for both input and output to functions, or really any other place.

lists A list consists of a sequence of values of the same type. For instance `[1; 3; 5]` is of type `int list`. Unlike arrays, list items are processed in that order, from left to right, and you can't easily jump to the middle of a list.

functions Functions themselves are values. Their type is often called an "arrow type". We will see it when we start creating functions.

arrays Arrays are less frequently used in OCAML. Their literal notation is similar to lists: `[| 1; 3; 5 |]`.

[5] http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html